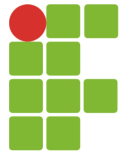
Árvores Sintáticas Abstratas

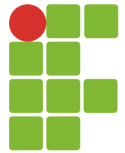
Compiladores

Prof. Daniel Saad Nogueira Nunes

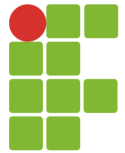
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

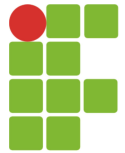


Sumário



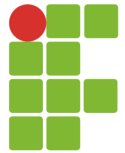
Introdução

- Em linguagens mais simples, compiladores podem ser escritos de modo a fazer a tradução em uma única etapa.
- **Tradução dirigida por sintaxe**.
- Contudo, em linguagens mais complexas, é necessário realizar utilizar mais de uma etapa.
- Para evitar a análise do programa em cada etapa, podemos construir uma estrutura de dados durante a análise sintática, conhecida como **árvore sintática abstrata**, que alimentará outras fases, como: a construção da tabela de símbolos, a análise semântica, e a produção de código.

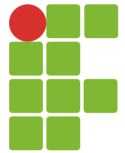


Introdução

- Nesta aula examinaremos como realizar a tradução dirigida pela sintaxe e como construir árvores sintáticas abstratas para dar suporte as outras etapas.

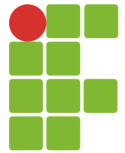


Sumário



Tradução dirigida por sintaxe

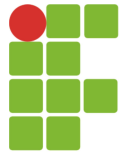
- Para alcançar a tradução dirigida por sintaxe, inserimos código no analisador sintático coadunado com as suas ações sintáticas.
- Dois conceitos importantes: **ações semânticas** e **valores semânticos**.



Tradução dirigida por sintaxe

Ações semânticas

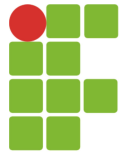
- Cada produção pode ter um código que será executado quando a produção é aplicada.
- Não há imposição sobre o que o código pode ou deve fazer.
- Esses códigos são conhecidos como **ações semânticas**, visto que seu escopo está além da sintaxe da gramática.
- Ações relacionadas ao **significado** de um programa.



Tradução dirigida por sintaxe

Valores semânticos

- Quando uma ação semântica é realizada para uma produção $\mathrm{A} \rightarrow \mathrm{X_1} \ldots \mathrm{X_m}$, valores estão relacionados com cada símbolo.
- O Valor associado a A é baseado nos valores atribuídos a $\mathrm{X_1} \ldots \mathrm{X_m}$.
- Para símbolos terminais, esses valores são obtidos na análise léxica.
- Por exemplo: o valor de um identificador é o seu nome.
- Para símbolos não-terminais, os valores são obtidos após a aplicação das regras.

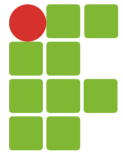


Tradução dirigida por sintaxe.

Considere a seguinte gramática:

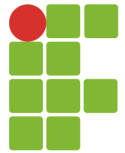
$$\begin{aligned}
&\text{Start} \rightarrow \text{E}\ \$ \\
&\text{E} \rightarrow \text{E}\ +\ \text{T} \\
&\text{E} \rightarrow \text{T} \\
&\text{T} \rightarrow \text{T}\ \cdot\ \text{T} \\
&\text{T} \rightarrow \text{num}
\end{aligned}$$

Em que num é um inteiro.



Tradução dirigida por sintaxe

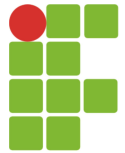
- Podemos escrever um analisador sintático LL(1)que calcula o resultado de uma expressão qualquer, como por exemplo $31 + 8 \cdot 50$, além é claro, de verificar que é uma expressão bem formada.



Tradução dirigida por sintaxe.

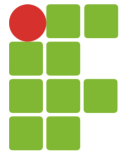
A gramática LL(1) equivalente é:

$$\text{Start} \rightarrow \text{E } \$$$
$$\text{E} \rightarrow \text{T } \text{E}'$$
$$\text{E}' \rightarrow \varepsilon$$
$$\text{E}' \rightarrow + \text{ T } \text{E}'$$
$$\text{T} \rightarrow \text{F } \text{T}'$$
$$\text{T}' \rightarrow \cdot \text{ F } \text{T}'$$
$$\text{T}' \rightarrow \varepsilon$$
$$\text{F} \rightarrow \text{num}$$



Tradução dirigita por sintaxe

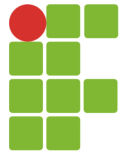
- Ao inserir as ações semânticas no analisador descendente recursivo obtemos o seguinte algoritmo.



Tradução dirigida por sintaxe

Algorithm 1: START()

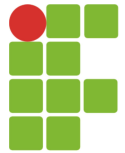
```
1 if( ts.PEEK(∈){num} )
2 |   val ← E()
3 |   ts.MATCH($)
4 |   return val
5 else
6 |   REPORT-SYNTAX-ERROR()
```



Tradução dirigida por sintaxe

Algorithm 2: E()

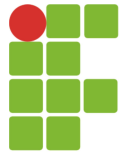
```
1 if( ts.PEEK() ∈ {num} )
2 |   val ← T()
3 |   val ← val + E'()
4 |   return val
5 else
6 |   REPORT-SYNTAX-ERROR()
```



Tradução dirigida por sintaxe

Algorithm 3: E'()

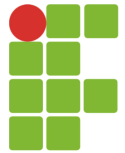
```
1 if( ts.PEEK() ∈ {$} )
2     return 0
3 else if( ts.PEEK() ∈ {+} )
4     ts.MATCH(+)
5     val ← T()
6     val_2 ← E'()
7     return val + val_2
8 else
9     REPORT-SYNTAX-ERROR()
```



Tradução dirigida por sintaxe

Algorithm 4: $\mathrm{T}()$

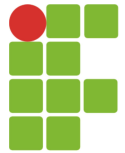
```
1 if( ts.PEEK() ∈ {num} )
2 |   val ← F()
3 |   val_2 ← T'()
4 |   return val · val_2
5 else
6 |   REPORT-SYNTAX-ERROR()
```



Tradução dirigida por sintaxe

Algorithm 5: T'()

```
1 if( ts.PEEK() ∈ {·} )
2     ts.MATCH(·)
3     val ← F()
4     val_2 ← T'()
5     return val · val_2
6 else if( ts.PEEK() ∈ {+, $} )
7     return 1
8 else
9     REPORT-SYNTAX-ERROR()
```

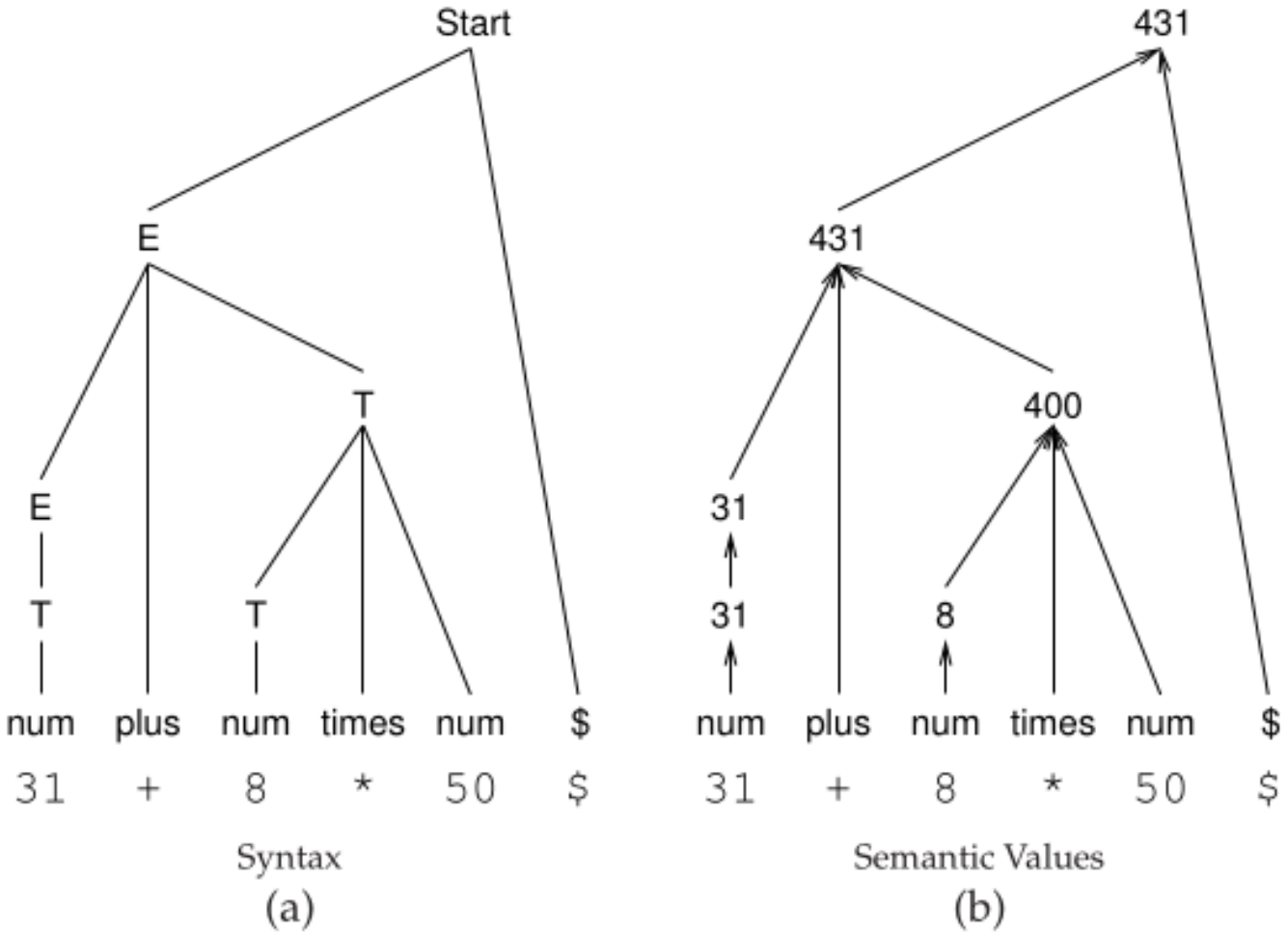


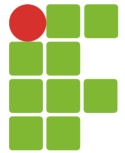
Tradução dirigida por sintaxe

Algorithm 6: F()

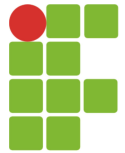
```
1 if( ts.PEEK() ∈ {num} )
2     val ← ts.VALUE()
3     ts.MATCH(num)
4     return val
5 else
6     REPORT-SYNTAX-ERROR()
```

Tradução dirigida por sintaxe



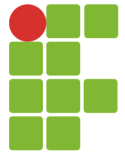


Sumário



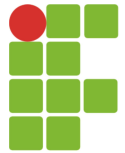
Árvores sintáticas abstratas

- Muitas tarefas de um compilador podem ser realizadas em uma única etapa através da tradução dirigida por sintaxe.
- Contudo, isso é desencorajado, visto que muitas funcionalidades acabam sendo incorporadas em um único componente do compilador, no caso, o analisador sintático.
- Tarefas como construção da tabela de símbolos, otimização de código, e análise semântica, merecem ter um tratamento separado.
- Consolidar todos esses componentes durante a análise sintática faz com que o código gerado seja difícil de ser mantido e estendido.



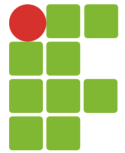
Árvores sintáticas abstratas

- A árvore sintática abstrata (AST) é uma estrutura de dados que pode ajudar nesse processo de separação de componentes.
- Ela é uma estrutura que é construída durante a análise sintática e utilizada pelos demais componentes.

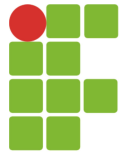


Árvores sintáticas abstratas

- ASTs devem ser, ao mesmo tempo, concisas e flexíveis.
- Devem ser simples, mas capazes de descrever o programa do seu ponto de vista sintático sem a perda de informações importantes.

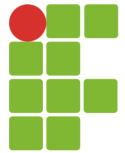


Sumário



Árvores sintáticas concretas e abstratas

- Árvores sintáticas **concretas** representam a entrada de acordo com a análise sintática.
- Contudo, elas possuem informações que muitas das vezes são desnecessárias para que a entrada seja realizada sem a ocorrência de ambiguidades, com por exemplo: delimitadores, palavras chaves e parênteses para garantir a ordem de avaliação de expressões.



Árvores sintáticas concretas e abstratas

Considere a seguinte gramática:

1 Start → Stmt $

2 Stmt → id assign E

3 | if lparen E rparen Stmt else fi

4 | if lparen E rparen Stmt fi

5 | while lparen E rparen do Stmt od

6 | begin Stmts end

7 Stmts → Stmts semi Stmt

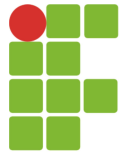
8 | Stmt

9 E → E plus T

10 | T

11 T → id

12 | num

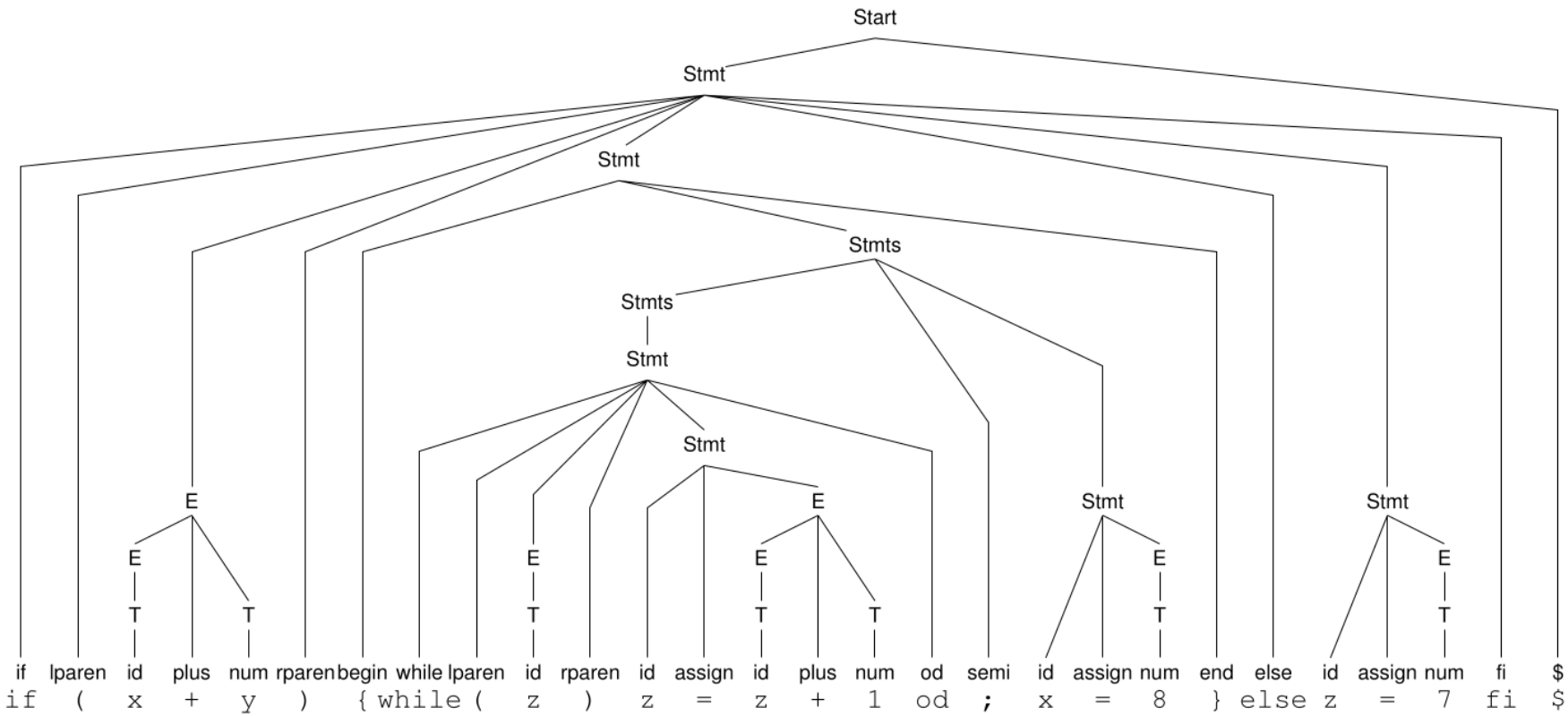


Árvores sintáticas concretas

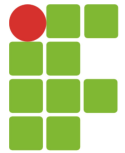
- A árvore sintática para a sequência de tokens

  ```
  if ( x + y ) { while ( z ) z = z+1
  od ; x = 8} else z = 7 fi $
  ```

  vem a seguir.



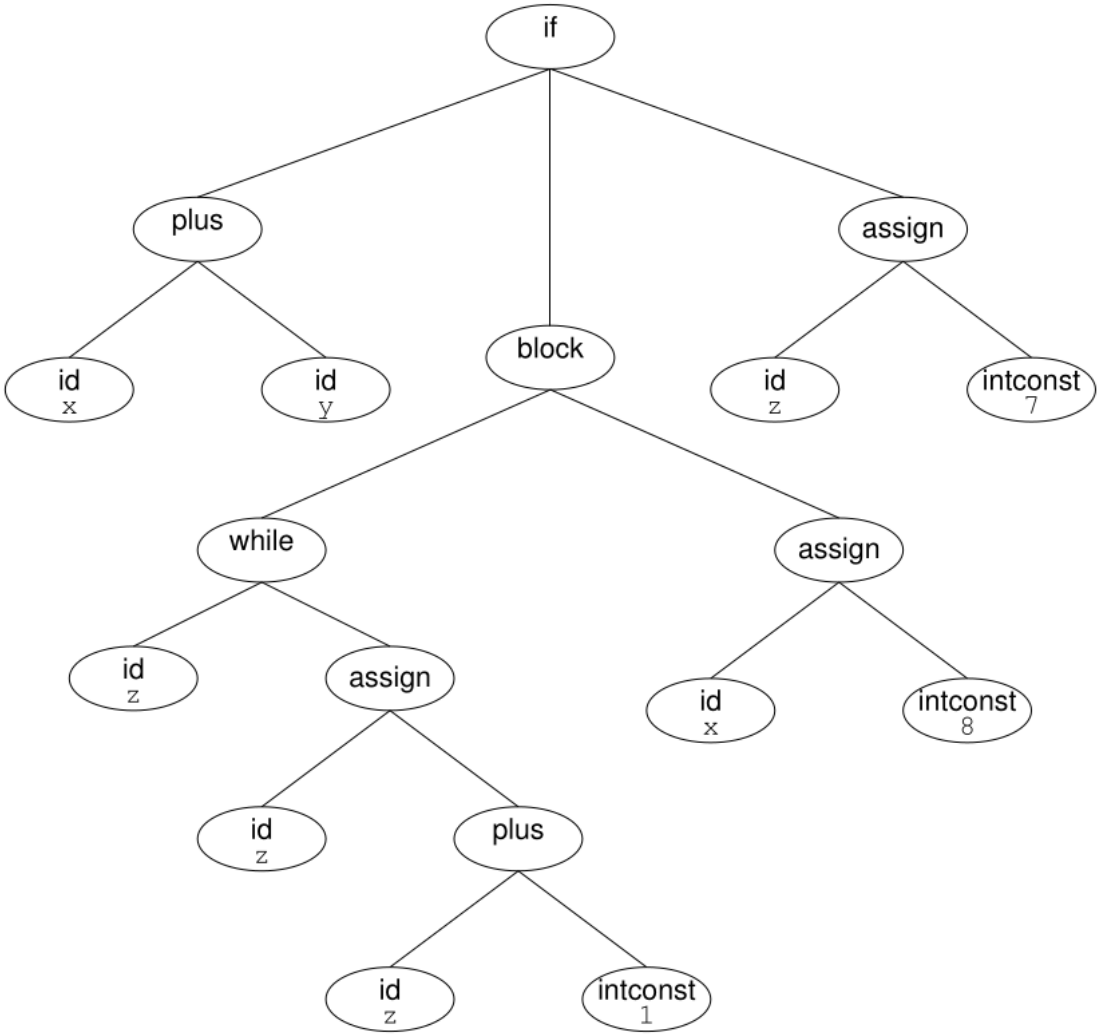
Árvores sintáticas concretas

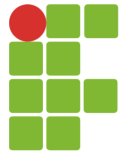


Árvores sintáticas abstras

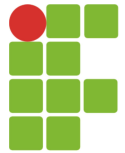
- A árvore sintática abstrata gerada durante o processo de análise sintática poderia eliminar algumas informações desnecessárias para as próximas etapas da compilação.
- A AST resultante vem a seguir.

Árvores sintáticas abstratas





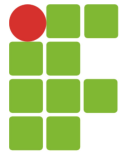
Sumário



EDs para ASTs

Para projetar uma ED eficiente para ASTs, devemos considerar que:

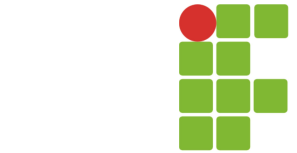
- As ASTs são construídas tipicamente de baixo para cima. Uma lista de irmãos é gerada que só depois serão adotados por um nó pai.
- A lista de irmãos é gerada tipicamente por regras recursivas. A ED deve possibilitar a adoção de irmãos de maneira simples através das extremidades da lista.
- Alguns nós da AST possui um número fixo de nós, como por exemplo um nó que representa uma soma, que tem dois filhos. Contudo, Algumas linguagens de programação podem requerer uma AST que suporte nós com vários filhos. A ED deve suportar uma aridade arbitrária das ASTs.



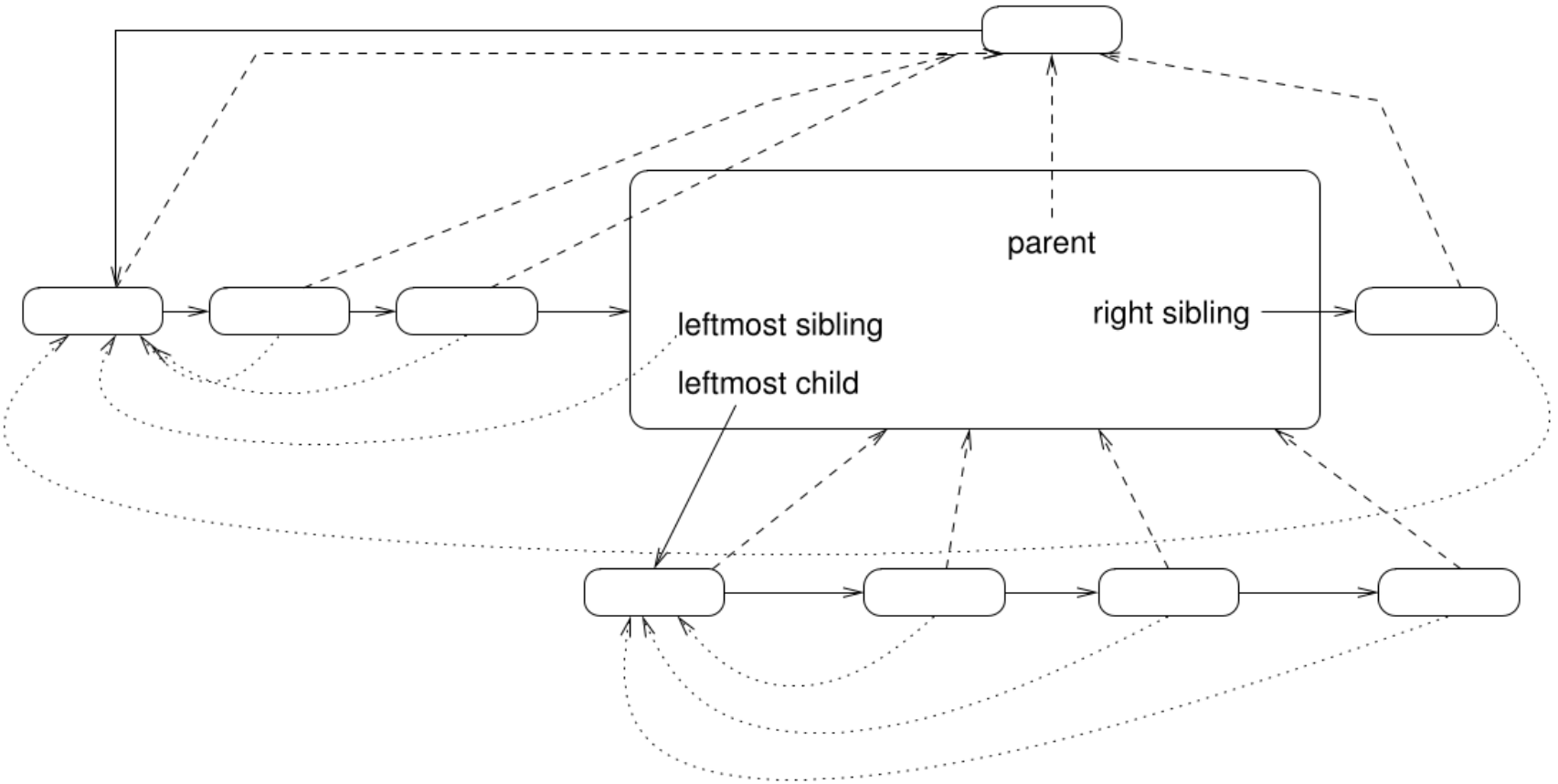
EDs para ASTs

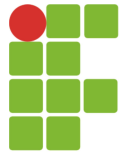
Para representar uma AST utilizaremos uma árvore com as seguintes propriedades.

- Os nós irmãos estão estruturados em uma lista encadeada simples, isto é, cada nó aponta para o seu próximo irmão. Além disso, cada nó aponta para a cabeça da lista, de modo a permitir a concatenação de listas (junção de irmãos) eficientemente.
- Cada nó possui um ponteiro para o seu pai.
- Cada nó possui um ponteiro para o seu filho mais à esquerda.



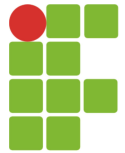
EDs para ASTs





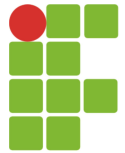
EDs para ASTs

- A função MAKE-NODE(t) segue o padrão factory e cria um nó que depende do tipo t. Por exemplo:
  - Ao receber um inteiro, cria-se um nó que representa uma constante inteira e possui um método para acessar o valor desse inteiro.
  - Ao receber um símbolo (identificador), cria um nó para representar o símbolo. Possui métodos para recuperar e modificar a entrada correspondente da tabela de símbolos.
  - Ao receber um operador, encapsula uma operação, como soma ou subtração, cujos detalhes são especificados por seus métodos.
  - Ao não receber nenhum parâmetro, cria um nó vazio, para representar a ausência de uma estrutura, como por exemplo uma estrutura condicional sem `senão`.



EDs para ASTs

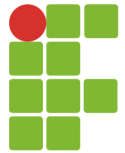
- A função $\mathrm{x.MAKE\text{-}SIBLINGS(y)}$ faz com que y seja irmão de x.
- Premissa: $x, y \neq \mathrm{NULL}$.
- Para isso, ele acha o irmão mais a direita de x e liga ele com o irmão mais à esquerda de y.
- Após isso, a lista dos irmãos de y é percorrida para:
  - Atribuir o pai de cada elemento ao pai de x.
  - Atribuir o irmão mais a esquerda de cada elemento ao irmão mais à esquerda de x.
- Ao final, retorna-se y, para possibilitar uma composição da função.



EDs para ASTs

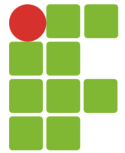
Algorithm 7: MAKE-SIBLINGS(y)

```
1  x ← this
2  xsibs ← x
3  while xsibs.next ≠ NULL do  xsibs ← xsibs.next
4  ysibs ← y.leftmostSib
5  xsibs.next ← ysibs
6  ysibs.leftmostSib ← xsibs.leftmostSib
7  ysibs.parent ← xsibs.parent
8  while ysibs.next ≠ NULL do
9      ysibs ← ysibs.next
10     ysibs.leftmostSib ← xsibs.leftmostSib
11     ysibs.parent ← xsibs.parent
12 return ysibs
```



EDs para ASTs

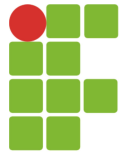
- A função $\mathrm{x.ADOPT(y)}$ faz com que x adote todos os irmãos de y, que passam a ter x como pai.
- Dois subcasos:
  - Se x tem filhos, então basta fazer com que o filho mais à esquerda de x vire irmão de y.
  - Caso contrário, é necessário percorrer a lista contendo y e seus irmãos e atribuir x como o pai. Além disso, x deve apontar para o irmão mais à esquerda de y.



EDs para ASTs

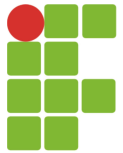
Algorithm 8: ADOPT(y)

```
1 x ← this
2 if( x.leftmostChild ≠ NULL )
3 |   x.leftmostChild.MAKE-SIBLINGS(y)
4 else
5 |   ysibs ← y.leftmostSib
6 |   x.leftmostChild ← ysibs
7 |   while ysibs ≠ NULL do
8 |   |   ysibs.parent ← x
9 |   |   ysibs ← ysibs.next
```



EDs para ASTs

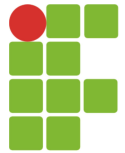
- A função $\textsc{make-family}(\mathrm{op}, \mathrm{kids})$ gera uma família com n nós, $\mathrm{kids}[0], \ldots, \mathrm{kids}[\mathrm{n}-1]$ tendo um nó de tipo op como pai.
- Pode ser implementada ao criar um nó node e adotar todos os irmãos $\mathrm{kids}[0], \ldots, \mathrm{kids}[\mathrm{n}-1]$.
- Ao final, node é retornado.



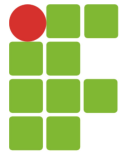
EDs para ASTs

Algorithm 9: MAKE-FAMILY(op, kids)

```
1 node ← MAKE-NODE(op)
2 sibs ← kids[0]
3 foreach i ∈ [1, n − 1] do
4     sibs.MAKE-SIBLINGS(kids[i])
5 node.ADOPT(sibs)
6 return node
```



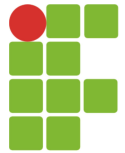
Sumário



Construção de ASTs

Já que uma árvore sintática está envolvida com muitas das etapas de um compilador, é importante se ater a alguns princípios:

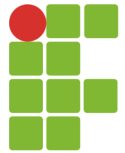
- Deve ser possível, a partir da AST, obter uma estrutura suficientemente similar ao programa representado pela AST, isto é, a AST deve ter informação o suficiente para representar a estrutura do programa subjacente.



Construção de ASTs

Já que uma árvore sintática está envolvida com muitas das etapas de um compilador, é importante se ater a alguns princípios:

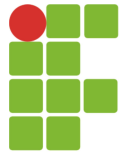
- Deve ser possível, a partir da AST, obter uma estrutura suficientemente similar ao programa representado pela AST, isto é, a AST deve ter informação o suficiente para representar a estrutura do programa subjacente.
- A implementação de uma AST deve estar desacoplada com a informação representa dentro dos nós das ASTs. Os nós devem encapsular a informação, que deve ser acessível por métodos para facilitar a interoperabilidade entre as fases de compilação.



Construção de ASTs

Já que uma árvore sintática está envolvida com muitas das etapas de um compilador, é importante se ater a alguns princípios:

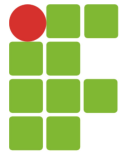
- Deve ser possível, a partir da AST, obter uma estrutura suficientemente similar ao programa representado pela AST, isto é, a AST deve ter informação o suficiente para representar a estrutura do programa subjacente.
- A implementação de uma AST deve estar desacoplada com a informação representa dentro dos nós das ASTs. Os nós devem encapsular a informação, que deve ser acessível por métodos para facilitar a interoperabilidade entre as fases de compilação.
- Como as fases da compilação veem os elementos de uma AST de formas diferentes, não há uma única hierarquia de classes para descrever a AST considerando todos os propósitos do compilador.



Construção de ASTs

Dada uma linguagem de programação L, o desenvolvimento de uma AST para L segue a seguinte estrutura, tipicamente:

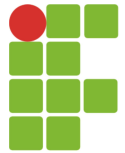
1. Uma gramática não ambígua para L é criada. Ela pode conter regras adicionais com o único propósite de desambiguação,



Construção de ASTs

Dada uma linguagem de programação L, o desenvolvimento de uma AST para L segue a seguinte estrutura, tipicamente:

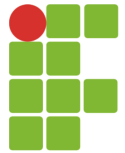
2. Uma AST para L é projetada. Normalmente, as regras extras criadas para fins de desambiguação, são descartadas. Símbolos semanticamente desnecessários, como `;` e `,` são usualmente descartados.



Construção de ASTs

Dada uma linguagem de programação L, o desenvolvimento de uma AST para L segue a seguinte estrutura, tipicamente:

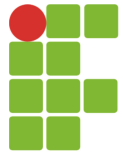
3. Ações semânticas são inseridas no analisador sintático para construir a AST. Essas ações semânticas podem usar os métodos que operam sobre a estrutura da AST para facilitar o processo.



Construção de ASTs

Dada uma linguagem de programação L, o desenvolvimento de uma AST para L segue a seguinte estrutura, tipicamente:

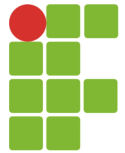
4 Estágios do compilador são projetados. Cada fase pode ter diferentes requerimentos da AST, seja pela estrutura ou pelo conteúdo.



Construção de ASTs

Para ilustrar a metodologia proposta, tome a especificação da seguinte linguagem de programação simples:

```
1   Start → Stmt $
2   Stmt → id assign E
3          |  if lparen E rparen Stmt else fi
4          |  if lparen E rparen Stmt fi
5          |  while lparen E rparen do Stmt od
6          |  begin Stmts end
7   Stmts → Stmts semi Stmt
8          |  Stmt
9   E → E plus T
10         |  T
11  T → id
```

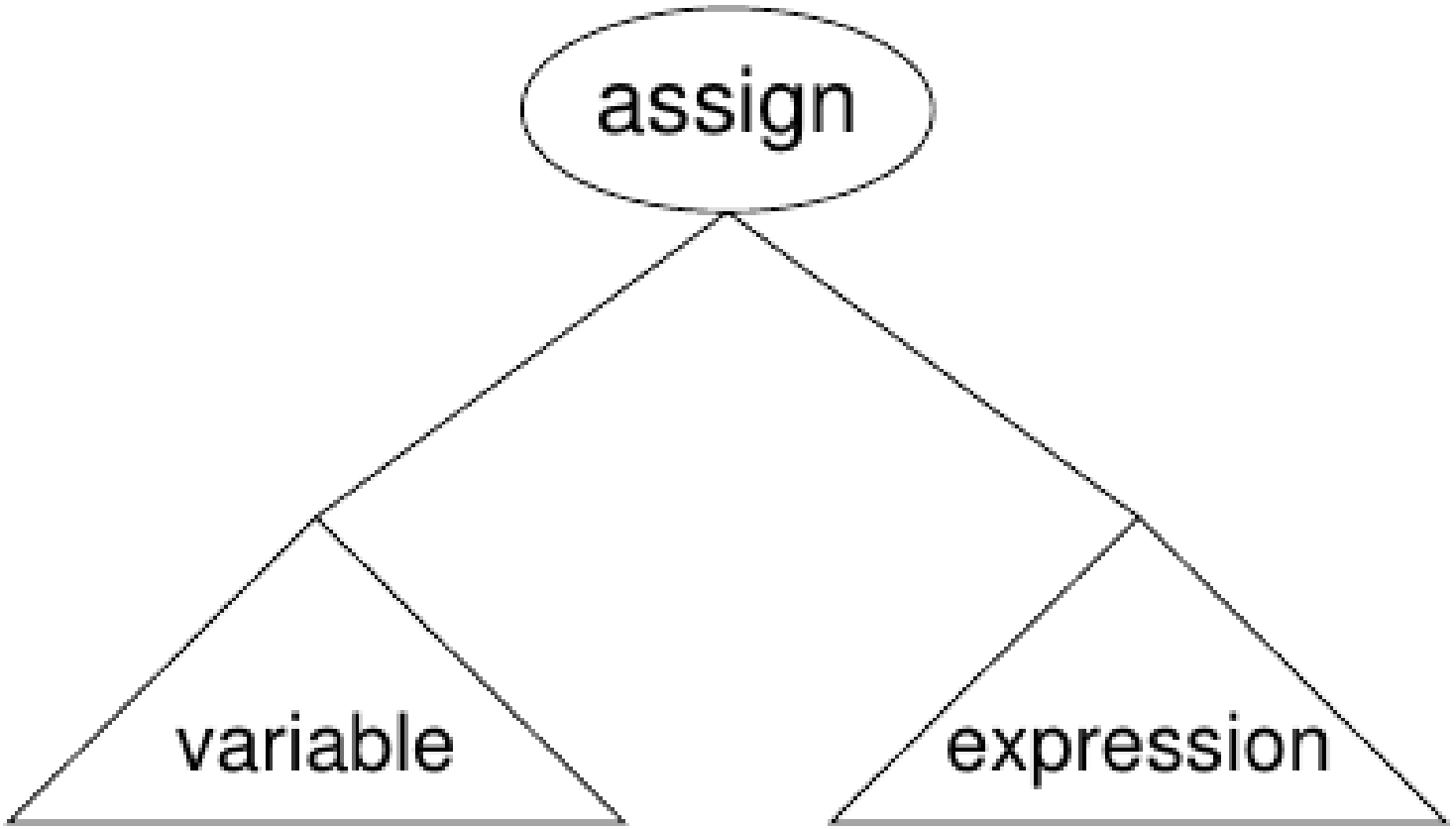


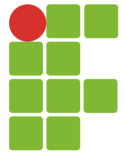
Construção de ASTs

Atribuição

A análise de tipos e a geração de códigos irão precisar de informações acerca da variável alvo da atribuição. A estrutura da AST será extremamente similar a da sua árvore concreta. O operador de atribuição será o nó pai do identificador, na subárvore da esquerda, e da expressão, na subárvore da direita.

Construção de ASTs: atribuição



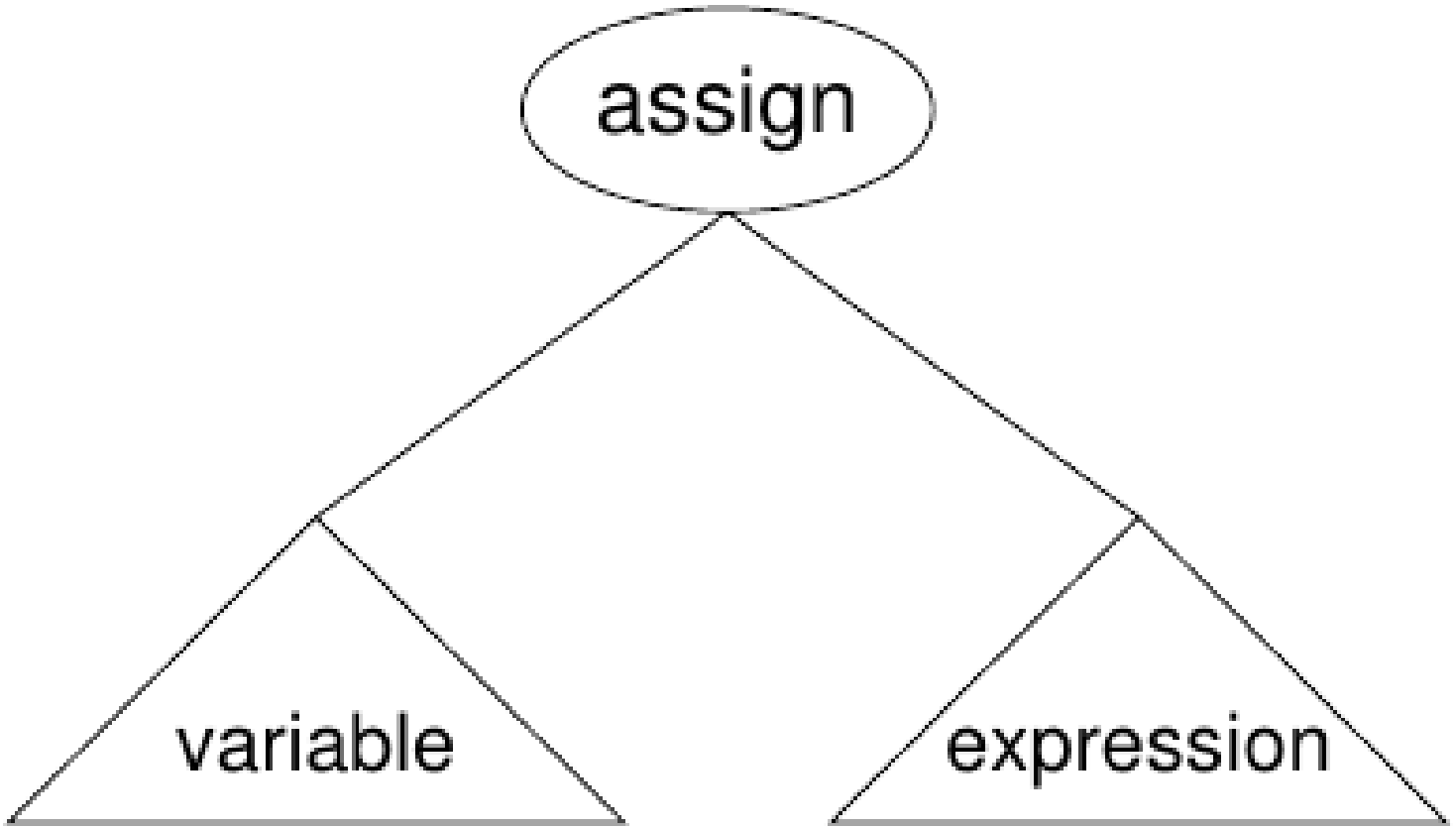


Construção de ASTs

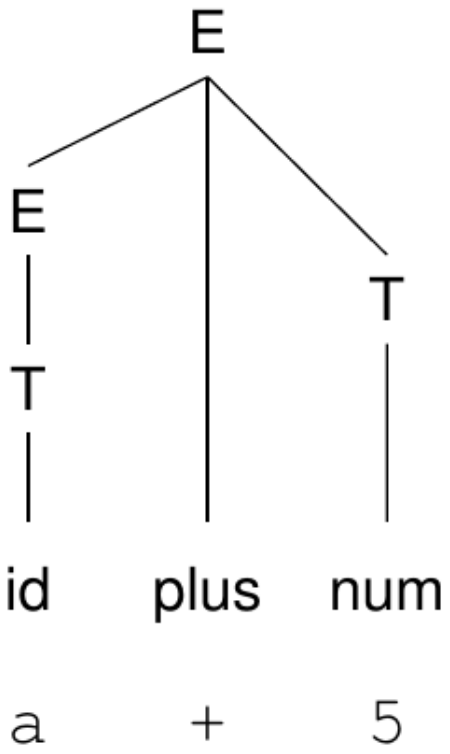
Operadores

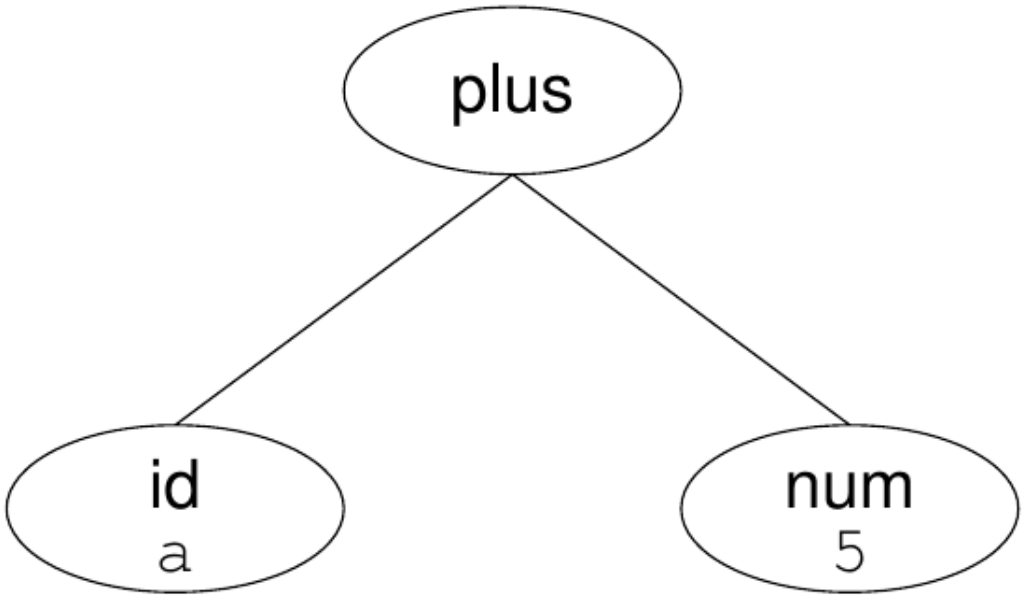
As regras cujo propósito servem para desambiguação podem ser descartadas. O nó pai será do tipo operador e as subárvores da esquerda e direita são as expressões aplicáveis ao operador.

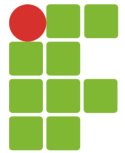
Construção de ASTs: Operadores



Construção de ASTs: Operadores





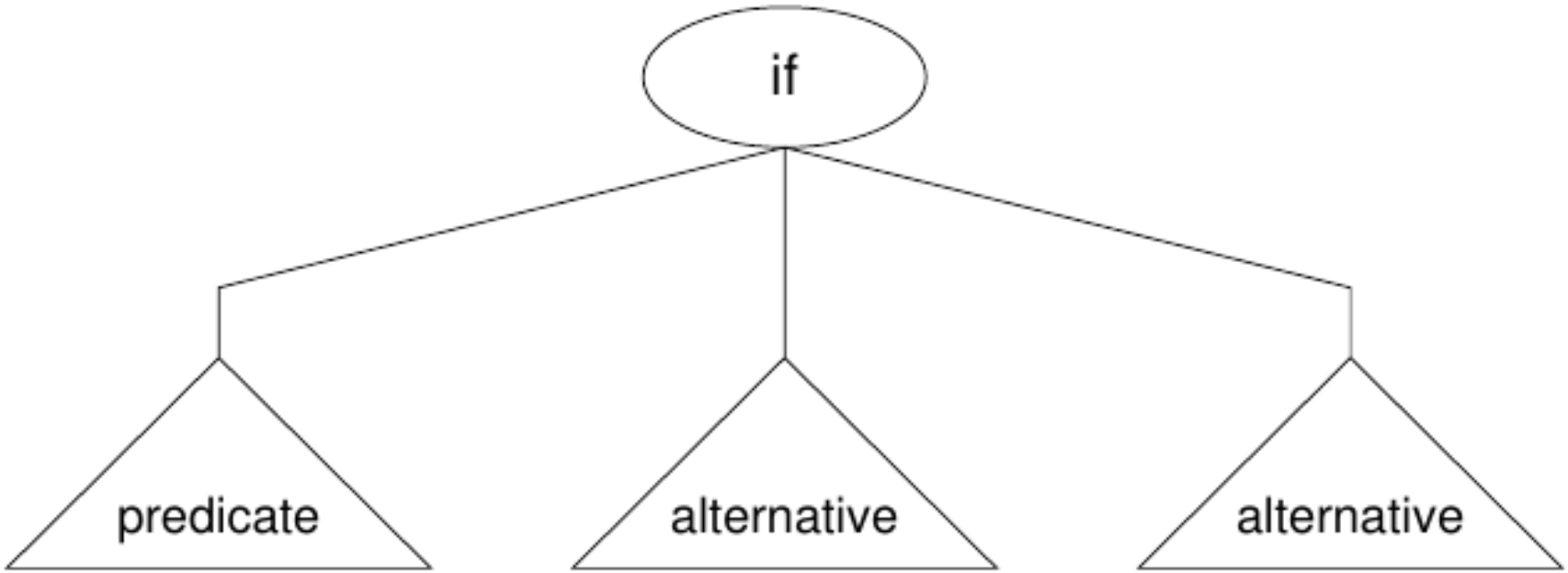


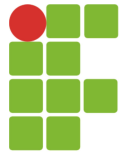
Construção de ASTs

Estruturas de decisão

Existem duas formas de estrutura de decisão presente na gramática. Uma com `else` e outra sem. Mas ambas podem ser vistas da mesma forma em uma AST, sendo a primeira implementada através de um nó vazio para representar um else. Alguns símbolos desnecessários, criados apenas como fins de desambiguação da gramática, podem ser removidos.

Construção de ASTs: estruturas de decisão

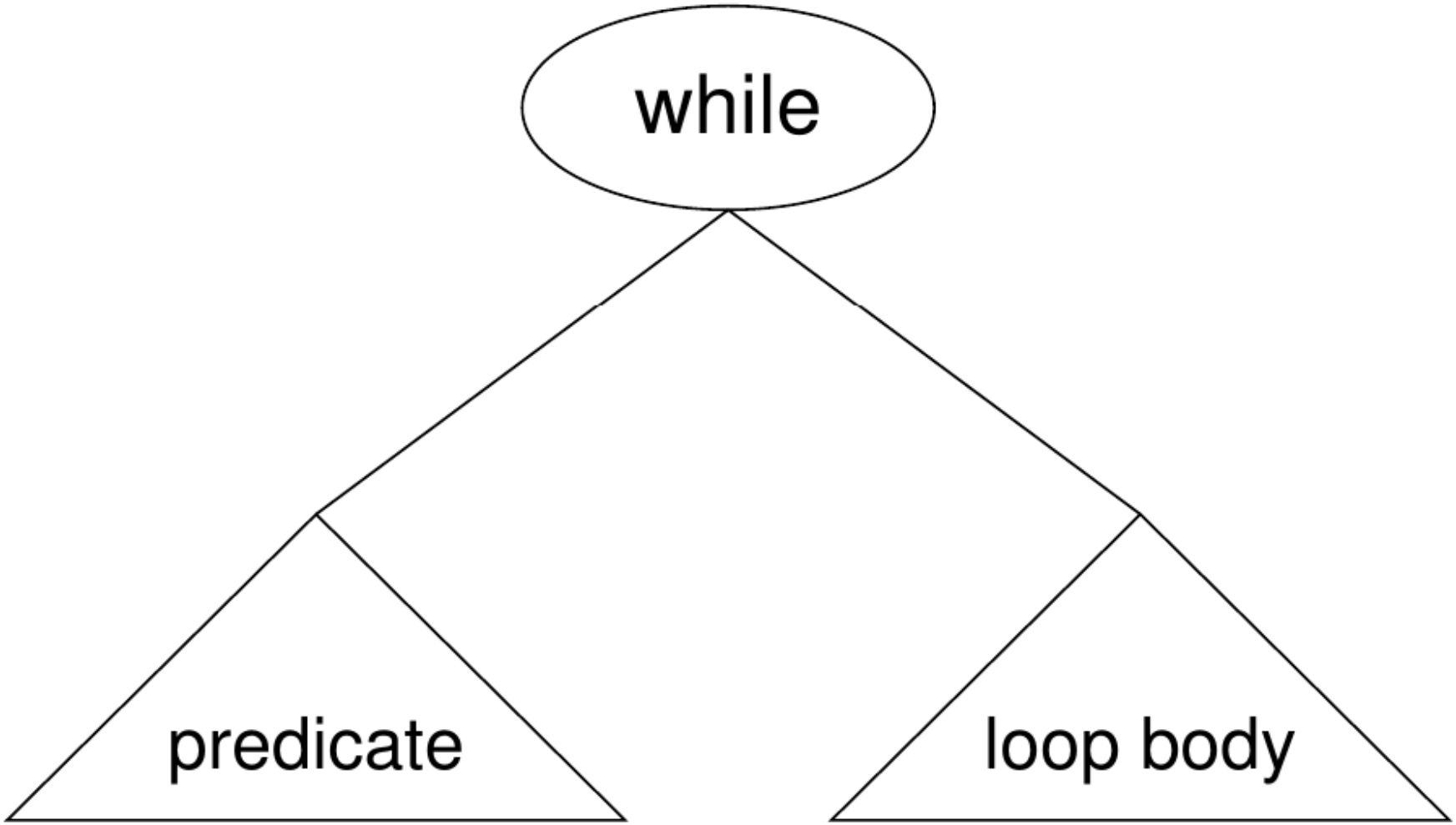




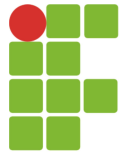
Construção de ASTs

Estruturas de repetição

Estruturas de repetição possuem dois elementos ligados ao nó pai: o predicado e o corpo do laço.



Construção de ASTs: estruturas de repetição

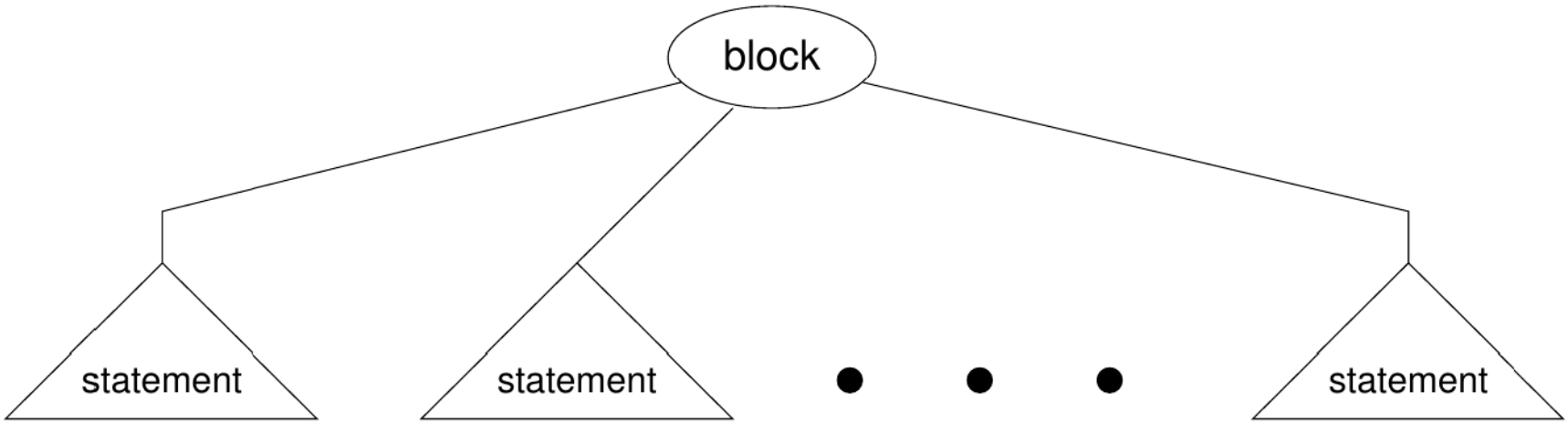


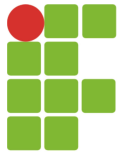
Construção de ASTs

Blocos de código

Diversas subárvores ligadas a um nó pai. Cada subárvore correspondendo a uma estrutura sintática.

Construção de ASTs: estruturas de repetição

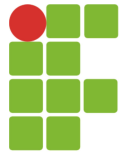




Construção de ASTs

Para construir a AST da linguagem, podemos adicionar as seguintes ações semânticas às regras:

$$1 \quad \text{Start} \rightarrow \text{Stmt}_{ast} \ \$ \quad \textbf{return} \ (ast)$$



Construção de ASTs

Para construir a AST da linguagem, podemos adicionar as seguintes ações semânticas às regras:

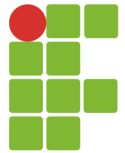
2 $\text{Stmt}_{result} \rightarrow \text{id}_{var}$ assign E_{expr}
$result \leftarrow \text{MAKEFAMILY}(\text{assign}, var, expr)$

3 | if lparen E_{p} rparen Stmt_{s} fi
$result \leftarrow \text{MAKEFAMILY}(\text{if}, p, s, \text{MAKENODE}())$

4 | if lparen E_{p} rparen Stmt_{s1} else Stmt_{s2} fi
$result \leftarrow \text{MAKEFAMILY}(\text{if}, p, s1, s2)$

5 | while lparen E_{p} rparen do Stmt_{s} od
$result \leftarrow \text{MAKEFAMILY}(\text{while}, p, s)$

6 | begin Stmts_{list} end
$result \leftarrow \text{MAKEFAMILY}(\text{block}, list)$

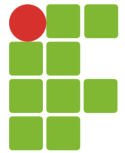


Construção de ASTs

Para construir a AST da linguagem, podemos adicionar as seguintes ações semânticas às regras:

7 $\text{Stmts}_{result} \rightarrow \text{Stmts}_{sofar}$ semi Stmt_{next}
$result \leftarrow sofar.\textsc{makeSiblings}(next)$

8 | Stmt_{first}
$result \leftarrow first$



Construção de ASTs

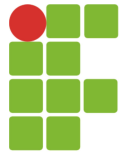
Para construir a AST da linguagem, podemos adicionar as seguintes ações semânticas às regras:

9 $E_{result} \rightarrow E_{e1}$ plus T_{e2}

$result \leftarrow$ MAKEFAMILY(plus, $e1, e2$)

10 | T_e

$result \leftarrow e$

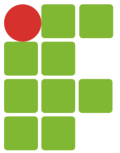


Construção de ASTs

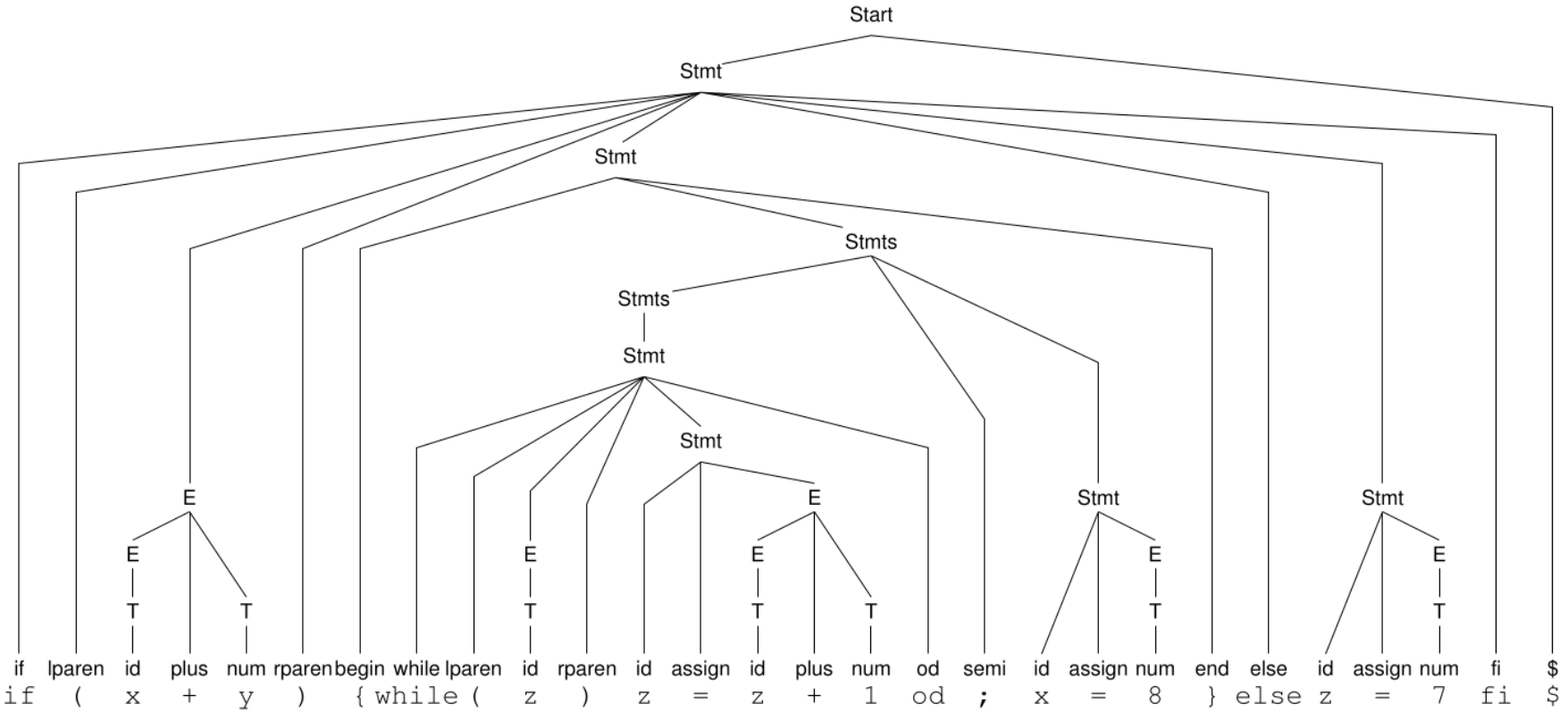
Para construir a AST da linguagem, podemos adicionar as seguintes ações semânticas às regras:

11 $\mathsf{T}_{result} \rightarrow \mathsf{id}_{var}$
$result \leftarrow \textsc{makeNode}(var)$

12 $\mid \mathsf{num}_{val}$
$result \leftarrow \textsc{makeNode}(val)$



Construção de ASTs



Construção de ASTs